Transformation Techniques for OCL Constraints

J. Cabot[a], E. Teniente[b]

[a]Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya,Rambla del Poblenou, 156 E08018 Barcelona

[b]Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Campus Nord, Ed. Omega, 132, E08034 Barcelona

Constraints play a key role in the definition of conceptual schemas. In the UML, constraints are usually specified by means of invariants written in the OCL. However, due to the high expressiveness of the OCL, the designer has different syntactic alternatives to express each constraint. The techniques presented in this paper assist the designer during the definition of the constraints by means of generating equivalent alternatives for the initially defined ones. Moreover, in the context of the MDA, transformations between these different alternatives are required as part of the PIM-to-PIM, PIM-to-PSM or PIM-to-code transformations of the original conceptual schema.

1. INTRODUCTION

Integrity constraints are a fundamental part in the definition of a conceptual schema (CS) [9]. In general, many constraints cannot be expressed using only the predefined constructs provided by the conceptual modeling language and require the use of a general-purpose (textual) sublanguage [6]. In the UML this is usually done by means of invariants written in the OCL [14]. Predefined (graphical) constraints can also be expressed in the OCL [8].

Due to the high expressiveness of the OCL, the designer has different syntactic possibilities to define an integrity constraint. For instance, given the CS in Figure 1, the constraint "the salary of an employee must be higher than the minimum salary of his/her department" may be defined as (among some other options):

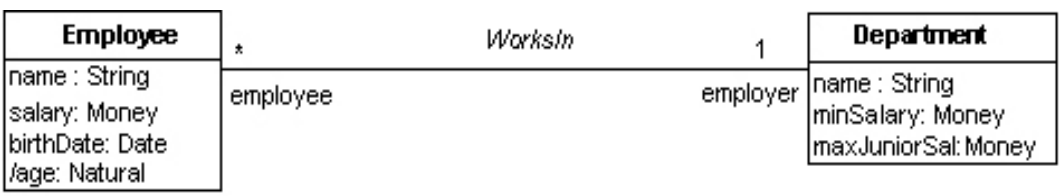

Figure 1. Example Conceptual Schema

1. **context** Department **inv**: self.employee -> forAll (e| e.salary>self.minSalary)
2. **context** Employee **inv**: self.salary>self.employer.minSalary
3. **context** Department **inv**: self.employee -> select(e| e.salary<=self.minSalary)->size()=0

Obviously, designers may not be aware of all different alternatives, and thus, they may just choose the one they care about at the moment of defining the constraint. Many times, this implies that the designer does not define the constraint in the *best* way, where the meaning of *best* may differ depending on the specific goal intended by the designer (for instance understandability or efficiency).

The ability of transforming the initially defined constraints into alternative equivalent representations is also important in the context of the MDA [16]. As an example, in PIM-to-PIM transformations, replacing the constraints with alternative representations is required as part of the refactoring operations at the model level [12]. In PIM-to-PSM or PIM-to-code transformations, aimed at generating a final implementation of the system directly derived from its specification, the generation of alternative representations may be necessary as an intermediate step before producing the actual implementation (for instance, when including the verification of the constraints as part of the operation contracts) or to produce a more efficient implementation [1].

There exist two different ways to generate an alternative representation for a given constraint: we can either replace the body of the constraint with an equivalent one (as it happens between constraints 1 and 3 of the previous example) or rewrite the constraint considering a different context type than the original one (as it happens with 1 and 2).

In this paper we propose several transformation techniques that allow to obtain a set of alternative constraint representations that are semantically equivalent to a given constraint. The replacement of the constraint body is handled by the definition of a set of equivalence rules between the different elements and constructs that may appear in the OCL expression defining the body of the constraint. On the other hand, the redefinition of the constraint using an alternative context is formalized as a path problem over a graph representing the CS. Using this graph we identify which entity types are candidates for acting as a new context type for the constraint and, then, we obtain all the possible redefinitions for each of them. Our proposal allows to generate all alternative redefinitions of the given constraint when using a different entity type as a context type but not all possible equivalent bodies for each alternative because of the huge number of equivalences among the different OCL constructs.

In contrast with the amount of research devoted to model transformations, redefinition of OCL expressions has received little attention in the past. In particular, [11] discusses the advantages of changing the context of a constraint but does not define which are the possible new contexts nor provides a method to generate such redefined constraints. Similarly, [4] proposes the context change as one of the possible refactorings to improve the specified OCL expressions but does not provide any method to automatically generate this context change. [7] provides some rules with the purpose of simplifying the constraints but the rules are not aimed at generating several alternative constraint definitions (and again, context changes are not addressed). [12] mentions context changes but restricts

them to associations with multiplicity 1 on both association ends. Hence, as far as we know, ours is the first method able to deal fully with transformations of OCL constraints.

The structure of the paper is as follows. The next section defines several equivalences between OCL expressions. Then, we propose transformation techniques to change the context of a constraint to a particular entity type (section 3) and we extend them to any type of the CS (section 4). Section 5 discusses some scenarios where the provided transformations are especially helpful. Section 6 presents our tool implementation. Finally, we give our conclusions and point out future work in Section 7.

2. EQUIVALENCES BETWEEN OCL EXPRESSIONS

As we said, one of the possible ways to generate an alternative representation for a certain constraint is to replace its body with an equivalent one. We achieve it by means of a set of equivalence rules between OCL expressions, which are described in this section. Each expression on the one side of the equivalence may be replaced with the expression on the other side to generate a new alternative body for the constraint. The set of rules is not exhaustive but it contains those equivalences we believe to be the most usual and/or useful ones according to our own experience.

As an example, assume we define an integrity constraint in the CS of Figure 1 to prevent junior employees (those with an age lower than 25) to earn more than the *maxJuniorSal* value defined for their department. This constraint could be defined in OCL as follows:

context *Department* ***inv*** $MaxSalary : Department.allInstances()\text{->}\ forAll(d|not\ d.employee-\text{>}\ select(e|e.age < 25)\text{->}\ exists(e|e.salary > d.maxJuniorSal))$

Applying the set of equivalences we propose, we could transform the expression defining the previous constraint into the equivalent one:

context *Department* ***inv*** $MaxSalary' : self.employee\text{->}\ forAll(e|e.age >= 25$ or $e.salary <= self.maxJuniorSal)$.

Note that the meaning of both constraints is exactly the same. However, in this case, the second expression is clearly simpler since the expression is shorter and it uses less operators.

Before applying the rules, we need to unfold the OCL expressions to maximize the number of applicable rules. We say that an OCL expression is unfolded when all references to derived elements, query operations and variables resulting from let expressions are replaced with their definition. To guarantee termination, we restrict recursive derived elements to be unfolded just once. Additionally, we assume that all implicit variables are made explicit.

We have specified the equivalence rules such that when applied in the left-right direction, the equivalences reduce the number of different operations that can appear in an OCL expression. For instance, a left-right application of our rules would allow removing the *exists* operation from any OCL expression. This facilitates the treatment of the OCL expressions by automatic methods since those methods do not need to take into account the full expressivity of the OCL.

On the other hand, when applying some of the rules in the right-left direction we may obtain more understandable expressions since some of them replace a sequence of several operations with a single operator. For instance, as shown in section 2.3, we may replace

a combination of *not* and *or* operators with the *implies* operator.

In general, designers will choose which rules to apply and in which direction depending on their intended goal.

Section 2.1 presents basic equivalence rules. Section 2.2 defines equivalences to remove the *allInstances* operation. Finally, section 2.3 provides equivalences to transform an OCL expression to conjunctive normal form (CNF). Equivalences in sections 2.1 and 2.3 may be applied to any OCL expression, including derivation rules and operation pre and postconditions; section 2.2 is specific for integrity constraints.

2.1. Basic equivalences

Tables 1, 2 and 3 define a list of basic equivalence rules. Most of these rules are based on the equivalences defined in the OCL standard itself [14]. We have grouped the equivalences by the type of expressions they affect (boolean, collection or iterator expressions). In the rules, the capital letters X, Y and Z represent arbitrary OCL expressions of the appropriate type. The letter o represents an arbitrary object. The expression $r_1 \dots r_n$ represents a (possibly empty) sequence of navigations.

Table 1
List of equivalences for boolean operators

X <> Y ↔ not X = Y	X = true ↔X
X = false ↔ not X	not false ↔ true
not true ↔ false	X and false ↔ false
X and true ↔ X	X or false ↔ X
X or true ↔ true	X>Y and X<=Y ↔ false
X>Y or X<=Y ↔ true	X>Y or X<Y ↔ X<>Y
not X>=Y ↔ X<Y	not X<Y ↔ X>=Y
not X<=Y ↔ X>Y	not X>Y ↔ X<=Y
not X=0 ↔ X>0 – when X evaluates to a natural value	X<=0 ↔ X=0 – when X evaluates to a natural value
X=Y ↔(X and Y) or (not X and not Y) – when X and Y are boolean expressions	

2.2. Removing the allInstances operation

AllInstances is a predefined feature on classes that gives as a result the set of all instances of the type that exist at the specific time when the expression is evaluated [14]. As an example, a constraint like "all employees must be older than 16" can be expressed as[1]:

context *Employee* ***inv*** *ValidAge: Employee.allInstances()->forAll (e| e.age>16)*

However, since constraints are assumed to be true for all instances of the context type (i.e. for all possible values of the *self* variable that represents any instance of the context type), the previous constraint could also have been specified as:

[1] *Type.allInstances()* can also be written as *Type::allInstances()*

Table 2
Equivalences for collection operators

X->includes(o) ↔ X->count(o)>0	X->excludes(o) ↔ X->count(o)=0
X->includesAll(Y) ↔ Y->forAll(y1|X->includes(y1))	X->excludesAll(Y) ↔ Y->forAll(y1| X->excludes(y1))
X->isEmpty()↔ X->size()=0	X->notEmpty()↔ X->size()>0
not X->isEmpty()↔ X->notEmpty()	not X->notEmpty()↔ X->isEmpty()
X->excluding(o)↔X-> −(Collection{o})	X->including(o) ↔ X->union(Collection{o})
X->union(Y).$r_1 \ldots r_n$->forAll(Z) ↔ X.$r_1 \ldots r_n$-> forAll(Z) and Y. $r_1 \ldots r_n$->forAll(Z)	X=Y ↔ X->includesAll(Y) and Y->includesAll(X) – when X and Y are collections of objects
X->last()↔ X->at(X->size())	X->first()↔ X->at(1)

Table 3
Equivalences for iterator expressions

X->exists(Y) ↔ X->select(Y)->size()>0	not X->exists(Y)↔X->forAll(not Y)
not X->forAll(Y) ↔ X->exists(not Y)	X->one(Y)↔X->select(Y)->size()=1
X->reject(Y) ↔ X->select(not Y)	X->select(Y)->size()=0 ↔ X->forAll(not Y)
X->select(Y)->forAll(Z) ↔ X->forAll(Y implies Z)	X->select(Y)->exists(Z) ↔ X->exists(Y and Z)
X->isUnique(Y) ↔ X->forAll(x1,x2 | x1<>x2 implies x1.Y <> x2.Y)	X->any(Y) ↔ X->select(Y)->asSequence()->first()
X->select(Y)->size()=X->size() ↔ X->forAll(Y)	X.$r_{1....}r_n$.Y.attr.Z ↔ X.$r_{1....}r_n$.Y->collect(attr).Z – where *attr* represents an attribute
X->forAll(Y) and X->forAll(Z) ↔ X->forAll(Y and Z)	X->forAll(v| Y [and|or] X->forAll(v2|Z)) ↔ X->forAll(v,v2| Y [and|or] Z))

context *Employee* ***inv*** *ValidAge': self.age>16*

We propose two equivalences to include/remove the *allInstances* operation in the expressions that define the body of an integrity constraint definition. They are applicable when the type over which *allInstances* is applied coincides with the context type (ct) of the constraint. They may not be applied if the constraint already contains any explicit or implicit reference to the *self* variable.

- *ct.allInstances()->forAll(v\| Y)* $\leftrightarrow$ *Y'*, where Y' is obtained by replacing all occurrences of v (the iterator variable) in Y with *self*. As an example, see the previous *ValidAge'* constraint.

- *ct.allInstances()->forAll(v1,v2..vn\| Y)* $\leftrightarrow$ *ct.allInstances()->forAll(v2..vn\| Y')* where Y' is obtained by means of replacing all the occurrences of v_1 in Y with *self*.

2.3. Transforming to conjunctive normal form

A logical formula is in conjunctive normal form (CNF) if it is a conjunction (sequence of ANDs) of several clauses, each of which is a disjunction (sequence of ORs) of one or more literals, possibly negated. Any logical formula can be translated into a CNF by applying a well-known set of rules.

We propose to apply that set of rules plus an additional rule to deal with the *if-then-else* construct to (de)normalize any boolean OCL expression in order to generate additional equivalent representations which may improve the results obtained by considering the rest of the rules alone. The rules we propose are the following:

1. Eliminate the *if-then-else* construct and the *implies* and *xor* operators:
   (a) X *implies* Y $\leftrightarrow$ *not* X *or* Y
   (b) *if* X *then* Y *else* Z $\leftrightarrow$ (X *implies* Y) and (*not* X *implies* Z) $\leftrightarrow$ (*not* X *or* Y) *and* (X *or* Z)
   (c) X *xor* Y $\leftrightarrow$ (X *or* Y) *and not* (X *and* Y) $\leftrightarrow$ (X *or* Y) *and* (*not* X *or not* Y)

2. Move *not* inwards until the negations be immediately before literals by repetitively using the laws:
   (a) *not* (*not* X) $\leftrightarrow$ X
   (b) DeMorgan's laws: *not* (X *or* Y) $\leftrightarrow$ *not* X *and not* Y
   not (X *and* Y) $\leftrightarrow$ *not* X *or not* Y

1. Repeatedly distribute *or* over *and* by means of:
   (a) X *or* (Y *and* Z) $\leftrightarrow$ (X *or* Y) *and* (X *or* Z)

2.4. Rule application

The different alternative representations of a given OCL expression are obtained by means of applying repetitively the previous equivalence rules. Beginning with an expression *exp*, the application of a rule r, transforms *exp* into an equivalent expression *exp'*. Then, any rule r' that can be applied over *exp* or over *exp'* generates a new alternative and so forth.

As an example, from the initial *MaxSalary* constraint definition we may obtain the alternative *MaxSalary'* representation by means of the following sequence of rules:

1. Initial representation:
   context *Department* ***inv*** *MaxSalary: Department.allInstances()->forAll(d| not d.employee->select(e|e.age<25)->exists(e|e.salary>d.maxJuniorSal))*

2. Removing the *allInstances* operation:
   context *Department* ***inv*** *MaxSalary: not self.employee->select(e |e.age < 25)->exists(e|e.salary>self.maxJuniorSal)*

3. Removing the *exists* iterator (rule *not X->exists(Y)* $\rightarrow$ *X->forAll(not Y)*):
   context *Department* ***inv*** *MaxSalary: self.employee->select(e|e.age<25)->forAll(e| not (e.salary>self.maxJuniorSal))*

4. Removing the *select* iterator (rule *X->select(Y)->forAll(Z)* $\rightarrow$ *X->forAll(Y implies Z)*):
   context *Department* ***inv*** *MaxSalary: self.employee->forAll(e| e.age<25 implies not (e.salary>self.maxJuniorSal))*

5. Transforming to CNF:
   context *Department* ***inv*** *MaxSalary: self.employee->forAll(e| not (e.age<25) or not (e.salary>self.maxJuniorSal))*

6. Removing the *not* operator (rules *not X<Y* $\rightarrow$ *X >= Y* and *not X>Y* $\rightarrow$ *X <= Y*):
   context *Department* ***inv*** *MaxSalary: self.employee->forAll(e| e.age >=25 or e.salary <= self.maxJuniorSal)*

Termination and confluence of the transformation process depends on the set of rules (and the direction in which they are applied) chosen by the designer. It is obvious that if the designer allows applying the same rule in both directions, then the generation process may enter into an infinite loop. To ensure termination we must avoid applying a rule when the rule will generate an alternative that has been already generated before. To ensure confluence we should define a total order regarding the selection of the rules to apply over an expression when several rules are applicable.

3. CHANGING THE CONTEXT TYPE OF A CONSTRAINT

In general, the designer may choose among several entity types to define the context of a particular constraint. Sometimes it is more useful to use a certain context, together with a corresponding constraint definition, instead of a different one.

Given two different context types ct_1 and ct_2 and a constraint c_1 defined over ct_1, we show in this section how to automatically obtain a constraint c_2 defined over ct_2 which is semantically equivalent to c_1. A constraint defines a condition that must be satisfied in all system states. More precisely, when defined in OCL, each constraint must be true for all instances of the context type where it is defined. We can therefore guarantee that two constraints c_1 and c_2 are semantically equivalent when the sets of instances taken into account by both constraints coincide and the condition to be evaluated over them is also the same.

In general, it may happen that several semantically equivalent constraints defined over ct_2 exist. Then, our transformation techniques generate all of them.

We assume in this section that the final context ct_2 is given by the designer (or by an external method). The next section generalizes the process by considering all possible new context types.

Changing the context type of a constraint makes only sense when the constraint is defined using a single instance of the context type (i.e. when the constraint body contains the *self* variable). Otherwise, i.e. when the constraint is defined with the *allInstances* operation, it is not worthy since its body will always be the same regardless the context chosen. Apart from that, the full expressivity of the OCL is allowed in the definition of the constraints.

In section 3.1, we assume that ct_2 is any entity type of the CS related with ct_1 through a sequence of associations. Afterwards, in section 3.2, we allow ct_2 to belong to the same taxonomy as ct_1. Both alternatives are not exclusive since ct_2 may belong to the same taxonomy as ct_1 and be also related with it.

3.1. Changing the Context between Related Entity Types

This section focuses on the transformation of a constraint c_1 with context ct_1 to a semantically equivalent constraint c_2 with context ct_2, where ct_1 and ct_2 are related through one or more sequence of associations that allow navigating between them.

According to one of the requirements to guarantee the semantic equivalence of c_2 and c_1, the context change from ct_1 to ct_2 is only possible when there exists at least one sequence of associations seq_{as} relating both types. Moreover, seq_{as} has to verify that $\text{set}_{ct1} = \text{set'}_{ct1}$; where set_{ct1} is the population of ct_1 (the set of instances that c_1 restricts) while set'_{ct1} is the set of instances of ct_1 obtained when navigating from the instances of ct_2 to ct_1 through seq_{as}. Otherwise, it is not possible to obtain a semantically equivalent constraint c_2 since it would not be possible to verify it over the same set of instances as c_1. In particular, the set of instances $\text{set}_{ct1} - \text{set'}_{ct1}$ would not be restricted by c_2.

We can determine whether $\text{set}_{ct1} = \text{set'}_{ct1}$ by studying the multiplicity of the associations included in seq_{as}.

Intuitively, if two entity types A and B are related through an association AB with the multiplicity *0..*:1..** (see Figure 2) it means that each instance of A is related at least to an instance of B. Thus, if we navigate from all instances of B to the related instances of A we necessarily obtain all A instances. Therefore, it is possible to change the context of a constraint defined in A from A to B. However, this is not the case from B to A because the minimum 0 multiplicity does not guarantee all instances of B to be related with instances of A.

For instance, the constraint "***context*** *A* ***inv:*** *self.a1 > 0*" may be translated to: "***context*** *B* ***inv:*** *self.a->forAll(a1>0)*". On the contrary, the constraint "***context*** *B* ***inv:*** *self.b1<5*" when translated to *A* (***context*** *A* ***inv:*** *self.b->forAll(b1<5)*) would not prevent that instances of B which are not related to A have a value in $b1$ lower than 5.

Then, we can state that $\text{set}_{ct1} = \text{set'}_{ct1}$ if the value of all minimum multiplicities of the roles used to navigate from ct_1 to ct_2 through the associations in seq_{as} is at least one. This guarantees that the navigation from ct_2 to ct_1 reaches all ct_1 instances. Following with the previous example, we can change the context of a constraint from A to B, A to C, B to C and C to B, but not from B to A or C to A.

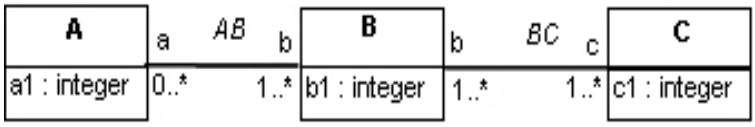

Figure 2. Example of an abstract conceptual schema

Depending on the specific body of the constraint we may be able to relax this multiplicity condition. When the body of c_1 permits to deduce that the constraint only affects those instances of ct_1 related with some instance of ct_2 we can use ct_2 as context of c_1. Roughly, this may happen when each literal appearing in the body of c_1 includes a navigation to ct_2. As an example, consider the *MaxSalary* constraint defined in section 2. Even though not all departments have employees assigned, the constraint only affects departments with employees (the others always satisfy the constraint). Thus, we can use *Employee* as an alternative context for the constraint.

Note that, for a given constraint, there may be several different sequences of associations from ct_1 to ct_2 that verify the previous condition. Each different sequence results in a different alternative representation of c_1.

We formalize the problem of changing the context between two related entity types as a path problem over a graph representing the CS. The next subsections explain how to create the graph, how to find the alternative paths and, for each one of them, how to obtain the new body of the constraint over the new context type.

3.1.1. Graph definition

The basic idea to represent the CS by means of a graph is to consider the entity types in the CS as vertices of the graph and the associations as edges between those vertices. Moreover, for our purposes, we want to obtain a graph that satisfies the following condition: if the graph contains a path from a vertex v_1 to a vertex v_2 then constraints defined over v_1 can be redefined using v_2 as a context type.

The graph must be a directed graph (digraph), since being able to change constraints from ct_1 to ct_2 (i.e. from the vertex representing ct_1 to the vertex representing ct_2) does not imply that we can also change constraints from ct_2 to ct_1, the context change is transitive but not symmetric. For instance, consider the graph of Figure 3, which is the

one obtained from the CS of Figure 2. The graph shows that constraints defined over A can also be expressed over B or over C. Constraints defined over B can be expressed over C but not over A. Constraints defined over Ccan be expressed over B.

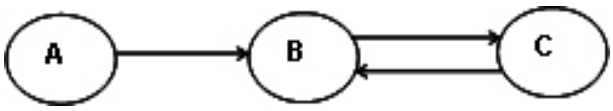

Figure 3. Example graph

Sometimes the graph may also be a multigraph since it may contain two or more edges with the same direction between a pair of vertices. This happens when the two corresponding entity types are related through more than one association.

According to those ideas, we build the graph G by means of the following rules:

1. All entity types, including reified ones (i.e. association classes), are vertices of G.

2. For each binary association between two entity types A and B, the edge $A \rightarrow B$ is included in G if the minimum multiplicity from A to B is at least one. The edge $B \rightarrow A$ is included when the minimum multiplicity from B to A is at least one.

3. Given a n-ary association As among a set of entity types $E_1 \dots E_n$ we add an edge from $E_i \rightarrow E_j$ if we can deduce, from the multiplicities of the roles in As, that the minimum multiplicity from E_i to E_j is at least one. Although these binary multiplicities are usually left unspecified in class diagrams, [13] shows that when the multiplicity of the role next to E_j is at least one, all the multiplicities from any E_i to E_j are at least one, and thus, the edge $E_i \rightarrow E_j$ is included in the graph.

4. For each vertex representing an association class AC, we add the edges $AC \rightarrow E_1$, $AC \rightarrow E_2, \dots, AC \rightarrow E_n$ where $E_1 \dots E_n$ are the participants of the association. We add these edges since an instance of an association class is always related to an instance of each participant type. We add the inverse edges depending on the multiplicities of the association. If AC is the reification of a binary association, we add $E_1 \rightarrow AC$ if $E_1 \rightarrow E_2$ exists (and conversely with E_2). Similarly, if the association is an n-ary association, we add $E_j \rightarrow AC$ if exists an E_i that verifies $E_j \rightarrow E_i$

5. Since subtypes inherit all the associations of their supertypes, for each edge $A \rightarrow B$ we add an edge $A_i \rightarrow B$ for each subtype A_i of A. Note that for edges of kind $B \rightarrow A$ we do not add $B \rightarrow A_i$ since the fact that each instance of B is related with an instance of A does not imply that it is also related with an instance of A_i.

The graph obtained with these rules is valid for any constraint. Then, if there is a path from ct_1 to ct_2 all constraints defined over ct_1 can be redefined using ct_2 as a context type.

As we have seen before, a context change from ct_1 to ct_2 may also be possible when the body of the original constraint only affects those instances of ct_1 related with instances of ct_2. To deal with these particular cases, we need to add to G some edges that are specific for certain constraints. For these reason, those edges are labelled with the name of a constraint and paths including them are only valid for changing the context of that particular constraint.

In Figure 4 we show the CS we will use as a running example in the rest of the paper. It specifies information about departments, their projects and employees and it includes the following six textual constraints. The first two are the constraints *MaxSalary* and *ValidAge* shown in section 2. The others ensure that departments with more than five employees are not managed by a freelance employee (*NotBossFreelance*), that all projects have at least two project managers (*AtLeastTwoProjectManagers*), that each employee assigned to a project finishes his contract after the due date of the project (*PossibleEmployee*) and that the number of hours per week that freelances work lies between 5 and 30 (*ValidNHours*).

- ***context*** *Department* ***inv*** *MaxSalary: self.employee->forAll(e | e.age >= 25 or e.salary <= self.maxJuniorSal)*
- ***context*** *Employee* ***inv*** *ValidAge: self.age>16*
- ***context*** *Department* ***inv*** *NotBossFreelance: self.employee->size()>5 implies not self.boss.oclIsTypeOf(Freelance)*
- ***context*** *Department* ***inv*** *AtLeastTwoProjectManagers: self.project->forAll(p | p.employee->select(e|e.category.name='PM')->size()>=2)*
- ***context*** *Project* ***inv*** *PossibleEmployeee: self.employee->forAll(e |e.expirationDate< self.dueDate)*
- ***context*** *Freelance* ***inv*** *ValidNHours: self.hoursWeek>=5 and self.hoursWeek<=30*

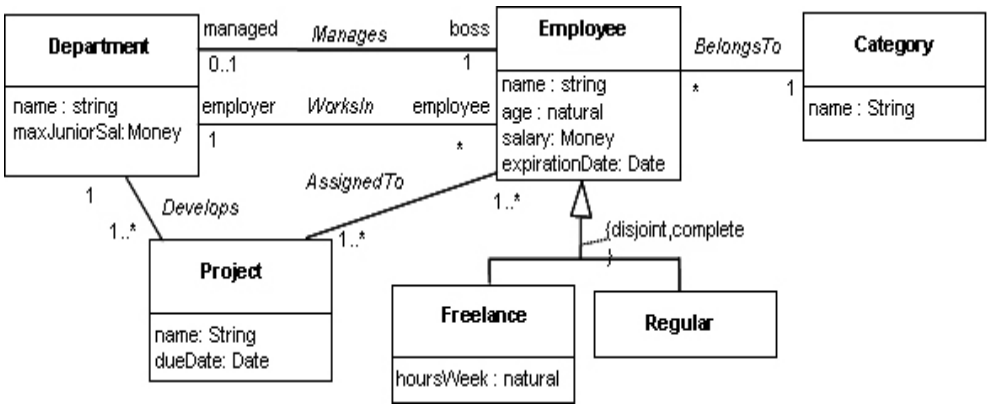

Figure 4. Conceptual schema used as a running example

Figure 5 shows the graph obtained from the previous CS. We can draw from it that constraints over *Project* may be redefined over *Employee, Department* and *Category*; constraints over *Employee* can be redefined over *Project*, *Department* and *Category*; constraints over *Category* can not be changed to any other context; etc.

The edge *WorksIn* from *Department* to *Employee* is labelled with the name of the constraint *MaxSalary* because this is the unique constraint that can be changed from *Department* to *Employee* using the association *WorksIn*.

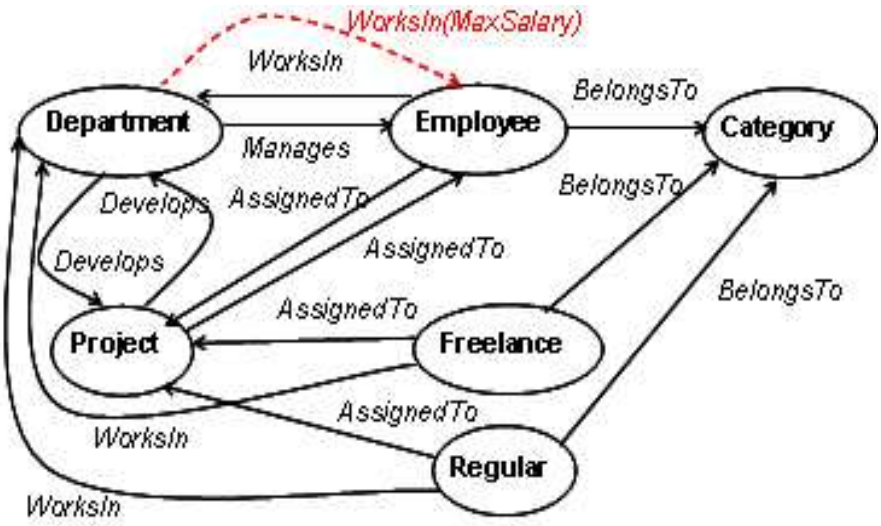

Figure 5. Graph of the conceptual schema

3.1.2. Computing alternative paths

Each different path from ct_1 to ct_2 represents a different way to express the original constraint c_1 in terms of the new context ct_2. To compute all alternative paths from ct_1 to ct_2 we have slightly adapted the depth-first graph searching procedure [10], using ct_1 as initial vertex and terminating the search only after all alternative paths reaching ct_2 have been generated. To avoid cycles, we do not consider as alternative paths those that contain repeated edges.

For instance, alternative paths from *Department* to *Employee* are the following: *Department-Manages-Employee* and *Department-Develops-Project-AssignedTo-Employee*. When looking for alternatives for the constraint *MaxSalary* we can also use the edge *WorksIn* from *Department* to *Employee*, and thus, there is an additional path: *Department-WorksIn-Employee*.

An alternative path may have repeated vertices. However, to simplify our presentation, we will not consider them in the rest of the paper.

3.1.3. Redefining the constraint over the new context type

Given a constraint c_1 with a body X defined over a context type ct_1, a context type ct_2 and a path p={$e_1, .., e_n$} (where $e_1..e_n$ are the edges linking the vertices {$ct_1, v_2, ..v_n, ct_2$}), the semantically equivalent constraint c_2 defined over ct_2 has the form:

context ct_2 ***inv*** c_2*: self.*$r_1.r_2...r_n$*->notEmpty() implies self.*$r_1.r_2...r_n$ *->forAll(v|X)*

where all occurrences of *self* in X have been replaced with v and $r_1..r_n$ are the roles that allow navigating from ct_2 to ct_1 using the associations appearing in p. Therefore, r_1 represents the navigation from ct_2 to v_n using the association e_n, r_2 the navigation from v_n to v_{n-1} using e_{n-1}, and, finally, r_n represents the navigation from v_2 to ct_1.

Intuitively, it can be seen that c_1 and c_2 are equivalent since both apply the same condition to the instances of ct_1 (the condition X) and apply it over the same set of instances (guaranteed by the graph definition process).

For instance, the constraint *MaxSalary* (***context*** *Department* ***inv****: self.employee-> forAll(e| e.age>=25 or e.salary<=self.maxJuniorSal)*) may be redefined over *Employee* because of the path p={*WorksIn*}. The redefined constraint *MaxSalary'* is:

context *Employee* ***inv****: self.employer->notEmpty() implies self.employer-> forAll(d| d.employee->forAll(e.age>=25 or e.salary<= d.maxJuniorSal))*

Since OCL does not define the navigation through n-ary associations, when e_i represents an n-ary association between v_{i+1} and v_i, we must navigate first from v_{i+1} to the corresponding association class and then from the association class to v_i.

Moreover, as ensured by the graph definition process, if an edge e_i links vertices v_{i+1} and v_i, there exists the corresponding association between the entity types E_{i+1} (represented by v_{i+1}) and E_i (represented by v_i) or between E_{i+1} and a supertype of E_i. In the latter case when navigating from E_{i+1} to E_i we need to add "*select(oclIsKindOf(subtype(*E_i*))*" (or "*any(oclIsKindOf(subtype(*E_i*))*" when the result must be a single object) to the corresponding r_i to ensure that only the instances of the subtype are retrieved by the navigation. For instance, the constraint *ValidNHours* can be translated from *Freelance* to *Category*. However, in the body of the resulting constraint, when navigating from *Category* to *Employee* we need to select just those employees that are freelances, since these are the only ones affected by the constraint. Then, the final body of *ValidNHours* when redefined over *Category* is the following:

self.employee->select(e| e.oclIsKindOf(Freelance))->forAll(f| f.oclAsType(Freelance).
hoursWeek >= 5 and f.oclAsType(Freelance).hoursWeek <=30)

We provide some rules to simplify the body of the new constraint c_2 (the variable X stands for an arbitrary boolean OCL expression).

1. *self.*$r_1...r_n$*->notEmpty()* $\rightarrow$ *true*, if the multiplicity of *self.*$r_1...r_n$ is at least one, i.e. if all the minimum multiplicities of $r_1 \ldots r_n$ are at least one. In this case, it is sure that the navigation will return a non-empty set and, thus, there is no need to apply the *notEmpty* operation.

2. *self.*$r_1 \ldots r_n$*->forAll(v|X)* $\rightarrow$ X (where all the occurrences of v in X are replaced with *self.*$r_1...r_n$), if the multiplicity of *self.*$r_1...r_n$ is at most one, i.e. if all the maximum multiplicities of $r_1...r_n$ are at most one. Then, the *forAll* iterator is no longer necessary.

3. *self.*$r_1 \ldots r_i.r_j \ldots r_n$*->forAll(X)* $\rightarrow$ *self.*$r_1 \ldots r_{i-1}.r_{j+1} \ldots r_n$*->forAll(X)*, when r_i and r_j are the two roles of the same binary association (see Figure 6). When the maximum multiplicity of r_j is one, the set of objects at r_j are the same than those at r_{i-1}, and thus, the navigations r_i and r_j are redundant (in this case the rule is applicable even if there is not a *forAll* iterator after r_n). Otherwise, we may have more objects at r_j, and, in general, this entails that these additional objects are not verified in the right hand expression of the rule. However, we can still apply the rule if the minimum multiplicity of all opposite roles from r_1 to r_{i-1} is at least one, since then, those objects must be related with a (different) instance of the context type, and thus, they will be checked when evaluating that instance. When r_i may have a zero minimum multiplicity, after the simplification we could be enforcing some objects not affected in the original constraint. Note that in such a case, the *notEmpty* clause of the general transformation rule will not be simplified by rule 1, and thus, we ensure that those objects will never be evaluated.

4. *self.*$r_1 ... r_n$*->forAll(v1,v2| X)* $\rightarrow$ *self.*$r_1 ... r_n$*->forAll(v2| X)* (where all occurrences of v_1 in X are replaced with *self*), if the type of the objects at r_n coincides with the type of the *self* variable and all the navigations from r_1 to r_n are redundant. This rule is similar to the rule to simplify the *allInstances* operation presented in section 2.2. We cannot completely simplify the *forAll* iterator since the constraint requires a comparison between an object of type t and a set of other objects of the same type t.

5. *self.*$r_1 ... r_i$*->forAll(v|* $v.r_j \ldots r_n$*->forAll(*v_2*|X))* $\rightarrow$ *self.*$r_1 \ldots r_i.r_j \ldots r_n$*->forAll(*$v_2$*|X)*, when X does not contain any reference to v. The two expressions are equivalent since in both we apply the condition X over the objects obtained at r_n. When X contains references to v they must be replaced with the expression $v_2.r_n' \ldots r_j'$ where $r_n'..r_j'$ represent the opposite roles of $r_n \ldots r_j$ (for instance, r_n' is the opposite of r_n). Note that when, the multiplicity of some r_k (where $j>=k<=n$) is greater than 1 then the left hand side must be replaced by the expression *self.*$r_1 \ldots r_i.r_j \ldots r_n$*->forAll(*$v_2$*|* $v_2.r_n' \ldots r_j'$*->forAll(*v_3*| X))* where references to v in the original constraint are replaced with v_3. This later case only makes sense when r_i and r_j are the two roles of the same association, which implies that the new expression can be simplified with rule 3 afterwards.

6. Given a reified entity type *RET* (see Figure 7): $X.ret.b.Y \rightarrow X.b.Y$. According to the OCL standard we can navigate to B either by accessing first the reified type or directly using the role b of B. In both cases we obtain the same set of instances.

7. Given a reified entity type *RET*: *context RET inv: self.a.b.*$r_1..r_n$*->forAll(X)* $\rightarrow$ *context RET inv: self.b.*$r_1..r_n$*->forAll(X)*. Even though, given an instance i of the *RET* type, the right hand side expression may verify less entities than the left hand expression (since $i.b$ may return less entities than $i.a.b$) those objects will be verified when evaluating other instances of *RET*.

All rules can be applied regardless the other subexpressions forming the constraint body except for rule 3 when the constraint body is a disjunction of literals following the pattern:

self.r_1*...*r_n*->forAll(X) or ... or self.*r_1*...*r_n*->forAll(Y)* (where $r_1 \ldots r_n$ represent exactly the same sequence of navigations in the all disjunctions). In this case, only the literal/s affected by the event for which the generated constraint is the appropriate alternative may be simplified. Assuming that the event is included in the X condition, the simplified constraint would be: *X or self.*r_1*...*r_n*->forAll(Y)*. Note that the body *"X or Y"* would not be a correct solution since the original constraint does not state that all entities at r_n must satisfy *X or Y*, it states that either all entities at r_n satisfy X or all entities satisfy *Y*. Then, if an event over an entity e makes that e evaluates X to false, we need to verify that at least all entities (and not just e) verify *Y*.

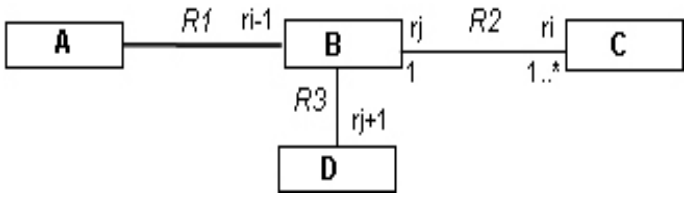

Figure 6. Abstract example schema for rule 3

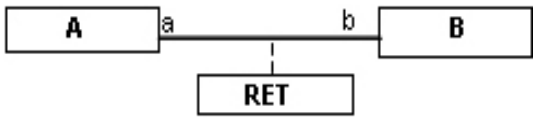

Figure 7. Example of a reified entity type

With the previous transformations, we can simplify the initially obtained *MaxSalary'* constraint as follows:

1. Initial representation after the context change:
   context *Employee* ***inv****: self.employer->notEmpty() implies self.employer->forAll(d| d.employee->forAll(e.age>=25 or e.salary<= d.maxJuniorSal))*
2. Removing the *notEmpty* operator (rule 1 plus the rules *true implies X* $\rightarrow$ *not true or X*, *not true or X* $\rightarrow$ *false or X* and *false or X* $\rightarrow$ *X*):
   context *Employee* ***inv****: self.employer-> forAll(d|d.employee->forAll(e.age>=25 or e.salary<= d.maxJuniorSal))*
3. Removing the first *forAll* (rule 2):
   context *Employee* ***inv****:*
   self.employer.employee->forAll(e.age>=25 or e.salary<= self.employer.maxJuniorSal)

4. Removing the redundant navigation (rule 3):
   context *Employee* ***inv****:*
   self->forAll(e.age>=25 or e.salary<= self.employer.maxJuniorSal)

5. Removing the *forAll* iterator (rule 2 again):
   context *Employee* ***inv****: self.age>=25 or self.salary<= self.employer.maxJuniorSal*

3.2. Changing the context within a taxonomy

Given a constraint c_1 defined over a context type ct_1, we are now interested to redefine c_1 using ct_2 as a context type, where ct_1 and ct_2 belong to the same taxonomy. This implies that either ct_1 is a subtype of ct_2, a supertype or both have a common supertype (ct_1 and ct_2 are sibling types).

When ct_1 is subtype of ct_2, the equivalent constraint c_2 defined over ct_2 has as a body: *self.oclIsTypeOf(ct_1) implies X*, where X is the body of c_1. In this way we ensure that c_2 is only applied over those instances that are instance of ct_1.

As an example, consider the constraint *ValidNHours*. If we want to move the constraint from *Freelance* to *Employee*, the new constraint would be:

context *Employee* ***inv*** *ValidNHours: self.oclIsTypeOf(Freelance) implies self.oclAsType(Freelance).hoursWeek>5 and self.oclAsType(Freelance).hoursWeek<30*

Note that, when accessing an attribute of the subtype, we need to use the *oclAsType* operator to do an explicit cast of the supertype variable.

If ct_1 is a supertype of ct_2, the new constraint c_2 is defined in ct_2 with exactly the same body as c_1. However, c_2 cannot replace c_1 since in general ct_1 may contain instances not appearing in ct_2. Thus, both constraints are not semantically equivalent[2]. If the set of generalization relationships between ct_1 and its direct subtypes is covering [15] (also called *complete*) c_1 can be replaced as long as we add a new constraint to each direct subtype of ct_1 with the same body as c_1. For instance, if we try to change the constraint *ValidAge* from *Employee* to *Freelance* we need to add also *ValidAge* to *Regular* to ensure that all employees have a valid age.

When ct_1 and ct_2 share a common supertype the new constraint c_2 can never replace c_1 since not all instances of ct_1 need to be instances of ct_2. As in the subtype case, the body of c_2 would be *self.oclIsTypeOf(ct_1) implies X*.

Before finalizing the context change to a new context entity type ct we can apply two simplification rules especially useful for this kind of transformations:

1. *self.oclIsTypeOf(ct)* $\rightarrow$ *true*
2. *self.oclAsType(ct).X* $\rightarrow$ *self.X*

4. COMPUTING ALL ALTERNTIVE CONTEXT CHANGES FOR A CONSTRAINT

To compute all possible context changes for constraint c_1, defined over ct_1, we need to consider all possible paths between ct_1 and every different type E appearing as a vertex of the graph.

[2]Except for those constraints where the body is already defined to apply only over the instances of the subtype ct_2.

In order to increase the number of possible alternatives, we may decide to reify some of the associations appearing in the original CS (especially those with multiplicities greater than one in all their roles). In this way, these reified types become vertices of the graph and turn out to be new candidate context types for the constraint.

In our example, the reification of the *AssignedTo* association implies the inclusion of a new vertex into the graph. Figure 8 shows the updated part of the graph of Figure 5, where new edges for the new vertex *AssignedTo* have been added according to the rules described in section 3.1.1.

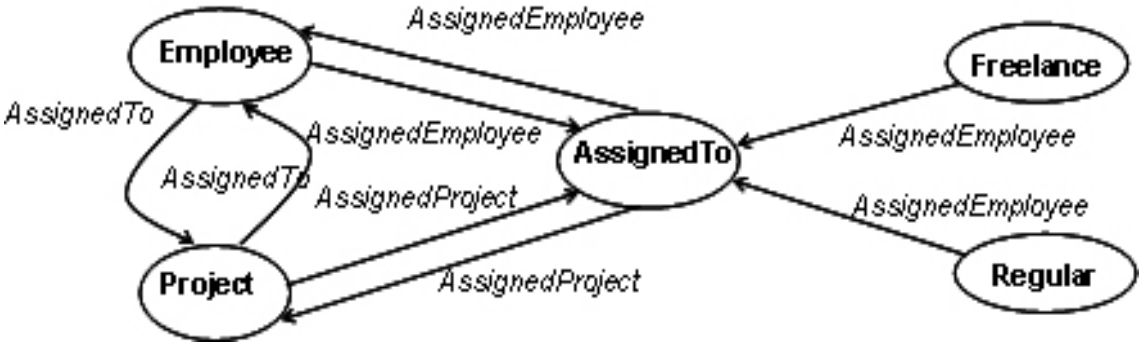

Figure 8. Updated part of the graph

As an example, we obtain sixteen different alternative representations for the constraint *MaxSalary* defined in Figure 4(one for every path between *Department* and the related types in the graph: *Employee*, *Project*, *Category* and *AssignedTo*). Table 4 shows the list of valid paths (column 2) for each possible final context (column 1).

Obviously, for each path we have a different alternative representation of the original constraint. The body of these new alternatives is computed as explained in section 3. Moreover, for each alternative we can apply the equivalences of section 2 to generate additional alternatives by means of changing the body of the obtained constraints. For paths including vertices representing entity types that participate in a taxonomy we must also consider the possible context changes along the taxonomy.

Nevertheless, we may reduce the search space by just considering the paths including only edges representing associations referred in the body of the original constraint. We can discard the other paths since alternatives obtained with them are surely more complex than the original one. Recall that any alternative constraint representation c_2 for a constraint c_1 obtained using the graph G initially presents a body consisting in a navigation (extracted from the path) from the context $ct_2$of c_2 to the context ct_1 of c_1 followed by the same body as c_1. Therefore, if no simplifications can be applied, c_2 is more complex than c_1 since its complexity may be regarded as that of c_1 plus that of the navigation from ct_2 to ct_1. Note that simplifications over c_2 can only be applied when the edges that form the path from ct_2 to ct_1 are also included in the body of c_1.

Therefore, to obtain the relevant alternative representations for a constraint c_1 it is

Table 4
Valid paths for *MaxSalary*

Context	**Path**
Employee	Department – *Manages* – Employee
	Department - *WorksIn* –Employee
	Department – *Develops* – Project – *AssignedTo* – Employee
	Department – *Develops* – Project – *AssignedProject* – AssignedTo – *AssignedEmployee* – Employee
Category	Department – *Manages* - Employee - *BelongsTo* – Category
	Department - *WorksIn* -Employee -*BelongsTo* – Category
	Department – *Develops* – Project – *AssignedTo* - Employee -*BelongsTo* – Category
	Department – *Develops* – Project – *AssignedProject* – AssignedTo – *AssignedEmployee* – Employee - *BelongsTo* – Category
Project	Department – *Develops* – Project
	Department – *Manages* – Employee – *AssignedTo* – Project
	Department – *Manages* – Employee – *AssignedEmployee* – AssignedTo – *AssignedProject* – Project
	Department – *WorksIn* – Employee – *AssignedTo* – Project
	Department – *WorksIn* – Employee – *AssignedEmployee* – AssignedTo – *AssignedProject* – Project
AssignedTo	Department – *Manages* – Employee – *AssignedEmployee* – AssignedTo
	Department - *WorksIn* –Employee– *AssignedEmployee* – AssignedTo
	Department – *Develops* – Project - *AssignedProject*– AssignedTo

enough to apply the previous algorithm over the graph G', subgraph of G, that contains the edges of G representing associations referenced in the body of c_1 along with their vertices and the vertices corresponding to the reified entity types of those edges (plus the edges between the reified type and the other entity types in G').

The subgraph G' corresponding to the constraint *MaxSalary* is shown in Figure 9. Using G' we reduce the number of alternative representations from sixteen to only one.

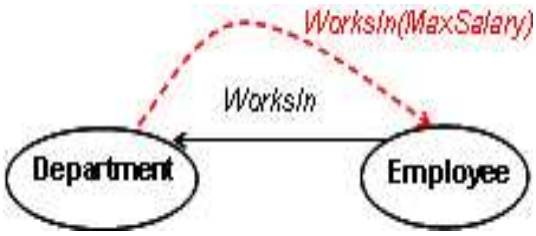

Figure 9. Subgraph for the constraint *MaxSalary*

According to this optimization, Table 4 summarizes the alternative representations (already simplified with the rules of section 3.1.3) for all constraints of our example. Note that for some constraints (as *ValidAge* and *ValidNHours*) the original representation is the only alternative. Notice also that *NotBossFreelance* can be defined over *Freelance* as a subtype of *Employee* because the body of the constraint can only be violated by *Freelance* instances.

For instance, the constraint *PossibleEmployee* over the reified type *AssignedTo* is first defined as:

context *AssignedTo* ***inv:*** *self.project->notEmpty() implies self.project->forAll(p|*
p.employee->forAll(e|e.expirationDate<self.project.dueDate)

and then simplified by means of removing the *notEmpty* operator (*self.project* has always a multiplicity value of 1), the first *forAll* (for the same reason) and, finally, applying the specific rules proposed for reified types.

5. APPLICATION SCENARIOS FOR THE CONSTRAINT TRANSFORMATION TECHNIQUES

The transformation techniques described in this paper are useful in several situations. First of all, at design time, we can use them to assist the designer in the definition of the constraints. With our proposal, designers may be aware of the different existing possibilities when defining each constraint, and thus, they can select the one they prefer. For students, the same transformation process could be useful to improve the learning of the OCL by means of showing them different alternatives for the constraints they define.

Moreover, the techniques are specially useful in several of the transformation scenarios defined in the context of the MDA [16]. In PIM-to-PIM transformations, several refactoring operations (see, [19] or [12] for examples) have been proposed to improve the design and structure of the models. OCL expressions may be affected by these refactorings and,

Table 5
Alternative representations for the example constraints (MS stands for MaxSalary, NFB stands for NotBossFreelance, ALTPM for AtLeastTwoProjectManagers, PE for PossibleEmployee, VA for ValidAge and VNH for ValidNHours)

Constraint	**Alternative representations**
MS	**context** Department **inv**: self.employee->forAll(e|e.age>=25 or e.salary<=self.maxJuniorSal)
	context Employee **inv**: self.age>=25 or self.salary<=self.employer.maxJuniorSal
NBF	**context** Department **inv**: self.employee->size()>5 implies not self.boss.oclIsTypeOf(Freelance)
	context Employee **inv**: self.managed.employee->size()<=5 or not self.oclIsTypeOf(Freelance)
	context Freelance **inv**: self.managed.employee->size()<=5
ALTPM	**context** Department **inv**: self.project->forAll(p| p.employee->select(e| e.category.name='PM')->size()>=2)
	context Project **inv**: self.employee->select(e| e.category.name='PM')->size()>=2
	context Employee **inv**: self.project->forAll(p| p.employee->select(e| e.category.name='PM')->size()>=2)
PE	**context** Project **inv**: self.employee->forAll(e| self.dueDate<e.expirationDate)
	context Employee **inv**: self.project->forAll(p| p.dueDate> self.expirationDate)
	context AssignedTo **inv**:self.project.dueDate>self.employee.expirationDate
VA	**context** Employee **inv**: self.age>16
VNH	**context** Freelance **inv**: self.hoursWeek>=5 and self.hoursWeek<=30

as a consequence, they need to be transformed to keep the consistency of the evolved model.

As a simple example, when removing a supertype (and moving its properties to the subtypes) we need to redefine the constraints having that type as a context type in terms of its subtypes. That may happen, for instance, if we remove the *Employee* type and move all its attributes and associations to *Freelance* and *Regular*. In this case we also need to change the context of the constraint *ValidAge* from *Employee* to both *Freelance* and *Regular*, and this can be done as explained in section 3.2.

Some other problems caused by refactoring operations are described in [12]. However, to avoid problems with the OCL expressions, in that proposal the refactoring operations are only allowed under certain strong restrictions which are not required in our approach. Therefore, our techniques can complement that method. We can also regard our transformation techniques as refactoring operations that improve comprensiveness of the OCL expressions. In this sense, we provide more powerful refactorings than the ones proposed in [4].

In PIM-to-PSM and PIM-to-code transformations, the goal is to derive (semi) automatically the final implementation of the system from the initial PIM specification. In this context, the simplicity of the defined constraints has a direct effect on the efficiency of the implementation. Therefore, our techniques can be used to increase the efficiency of the final system by generating equivalent but more efficient constraints than the original ones written by the designer (see [1] for full details of the process).

Additionally, when the final technology platform does not offer a predefined mechanism to implement the constraints (as it happens with object-oriented languages), the constraints need to be implemented using alternative constructs. Typically, constraints are included in the contracts of the system operations that may induce a constraint violation. For instance, in an operation *RaiseSalary* defined in the entity type *Employee*, we need to check that the new salary does not violate the *MaxSalary* constraint. In order to be able to easily include its verification in the contract of *RaiseSalary*, the constraint needs to be defined in terms of the employee instances, and thus, we need, first, to transform *MaxSalary* from its original version (using *Department* as a context type) to a new version defined using *Employee* as a context type.

Our transformation techniques may be useful in schema validation as well, when comparing a set of constraints in search of redundancies among them. For instance, it could help in the detection that two or more constraints are equivalent by means of redefining them over the same context type, processing their body with the rules of section 2 in the left-right direction and, finally, comparing the final resulting expressions. In some cases, just by comparing the final body we could determine if both constraints are equivalent or not (for instance, our techniques are able to detect that the three versions of *MinSalary* shown in the introduction are redundant), but, in any case, our transformation may help existing model checkers to provide better results.

As an additional benefit, preprocessing OCL expressions by means of the equivalence rules of section 2 in the left-right direction facilitates the creation of methods and tools to process OCL expressions (as, for instance, tools to implement transformations defined using OCL or OCL-like languages as in the QVT standard [18]). Thanks to our preprocessing, those methods do not need to address the full expressivity of the OCL, they just

need to consider the simplified grammar resulting from our rules (for instance, they may forget about the iterators *one*, *isUnique*, *reject*, ...). This reduces their complexity.

Similarly, our transformations are also helpful to extend the set of expressions handled for already existing tools. As an example, current OCL-to-Java tools (i.e. tools that translate OCL expressions to Java code, see [2] for a survey) cannot handle expressions that include the *allInstances* operator. Some of those expressions could be simplified with the rules of section 2.2, and thus, they could be processed by the tools. Moreover, some of the tools require constraints to be defined over a type t in order to properly verify them after the execution of operations defined in t. Here, our context change transformations may be in charge of providing an adequate representation of the constraints using t as a context type when possible.

6. TOOL IMPLEMENTATION

We have implemented a prototype tool [3] for the transformation techniques presented in this paper. Given an XMI file [17] representing a CS and a set of OCL integrity constraints in textual form (parsed using the Dresden OCL toolkit [5]), our tool generates all possible context changes for those constraints. The generated constraints are shown to the designer and may be stored, if required, in an output text file.

As a first step, the input constraints are preprocessed by means of applying the equivalence rules of section 2 in the left to right direction. Since each rule has been implemented in a separate Java class, our tool could be easily extended with the inclusion or the removal of new equivalence rules according to the designers' interest.

Then, and according to the input CS, the graph representing the CS is created and all possible paths are computed. As an example, we show in Figure 10 the results of processing our running example with the tool.

Finally, the user may select some (or all) the constraints in order to generate their alternative representations following the previous paths and the taxonomic relationships. The generated constraints are simplified (with the equivalences of section 3.1.3) and shown to the user. Along with the final constraints, the tool also provides information about the path and the rules applied to obtain them (see Figure 11).

7. CONCLUSIONS AND FURTHER WORK

We have proposed several transformation techniques that allow obtaining a set of semantically equivalent representations for a given OCL constraint. The techniques consider both changes in the body of the constraint as well as the possibility of redefining the constraint using as a context a different entity type of the conceptual schema. As far as we know, ours is the first proposal able to generate all alternative representations of a given integrity constraint in terms of different context types.

Although we have focused on integrity constraints, most of the equivalences of section 2 are useful for any kind of OCL expressions while the context changes of section 3 are partially applicable to pre and postcondition expressions (which, in fact, are represented as stereotyped constraints in the UML metamodel) as well.

The main part of our proposal is formalized as a path problem over a graph representing the conceptual schema. The graph is created in such a way that every path between two

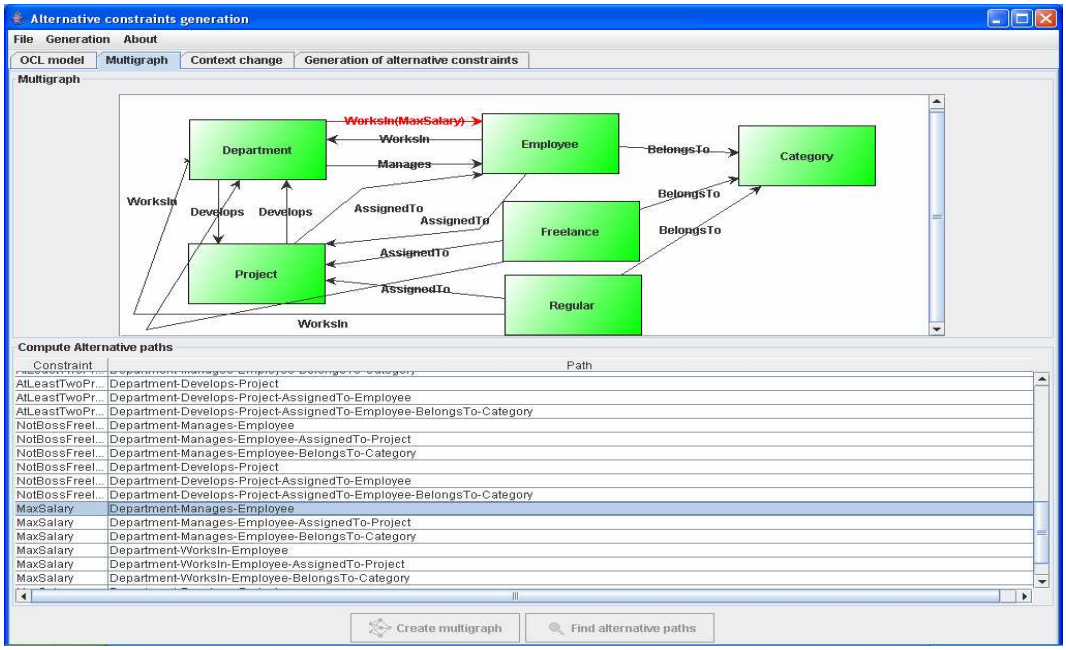

Figure 10. Graph and paths computed by the tool

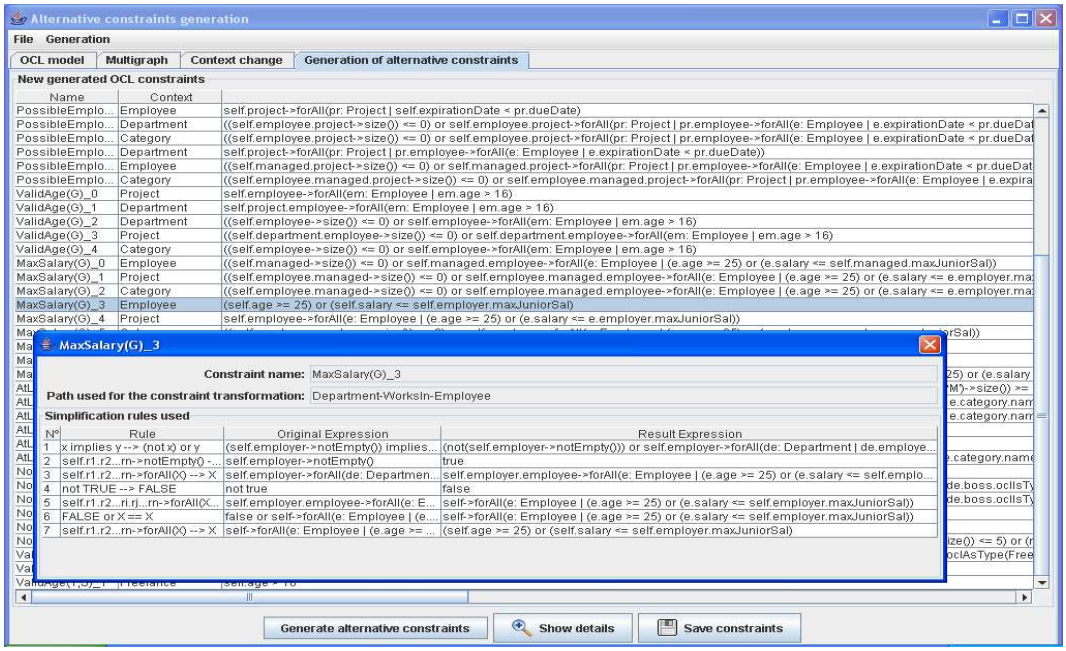

Figure 11. Constraints generated by the tool

vertices corresponds to a different alternative to represent the set of constraints defined over the first vertex (i.e. over the entity type represented by the vertex) by using the second one as a context. Using this graph we are able to compute the different alternative representations of a given OCL constraint.

Thanks to this generation, designers are aware of the different possibilities they have when defining a constraint and may select the best one according to their particular interest (for instance readability, understandability or efficiency). Our techniques are also useful in several transformation scenarios, as, for instance, the implementation of refactoring operations over the CS or the automatic code generation of the constraints in the final technology platform.

Further research may involve looking for additional useful equivalences that may improve further the results of our techniques (for instance, integrating some of the equivalences presented in [7]). Moreover, in many situations, the designer is interested in finding the *best* representation for a constraint among all alternative representations. We would like to define a set of complexity models that, depending on the designers' goal, allow obtaining automatically the *best* representation of a given constraint.

8. ACKNOWLEDGEMENTS

We would like to thank J. Conesa, D. Costal, C. Gómez, A. Olivé, A. Queralt, R. Raventós and M. R. Sancho for their many useful comments in the preparation of this paper. Special thanks to R. Solana for his contribution with the tool implementation. This work has been partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIN2005-06053.

REFERENCES

1. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering, LNCS, 4001 (2006) 81-95
2. Cabot, J., Teniente, E.: Constraint Support in MDA tools: a Survey. In: Proc. 2nd European Conference on Model Driven Architecture, LNCS, 4066 (2006) 256-267
3. Cabot, J., Teniente, E. A tool for the transformation of OCL constraints.[Online]. Available: www.lsi.upc.edu/∼jcabot/research/OCLTransformations
4. Correa, A., Werner, C.: Refactoring object constraint language specifications. Software and Systems Modeling 6 (2006)
5. Dresden. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.html
6. Embley, D. W., Barry, D. K., Woodfield, S.: Object-Oriented Systems Analysis. A Model-Driven Approach. Yourdon Press Computing Series. Yourdon (1992)
7. Giese, M., Larsson, D.: Simplifying Transformations of OCL Constraints. In: Proc. 8th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'05), LNCS, 3713 (2005) 309-323
8. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: A. Clark and J. Warmer, (eds.): Object Modeling with the OCL. Springer-Verlag (2002) 85-114
9. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)

10. Jungnickel, D.: Graphs, networks and algorithms. Springer-Verlag (1999)
11. Ledru, Y., Dupuy-Chessa, S., Fadil, H.: Towards computer-aided design of OCL constraints. In: CAiSE'04 Workshops Proceedings, Vol. 1. Faculty of Computer Science and Information Technology, Riga Technical University, Riga, Latvia (2004) 329-338
12. Markovic, S., Baar, T.: Refactoring OCL Annotated UML Class Diagrams. In: Proc. 8th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'05), Lecture Notes in Computer Science, 3713 (2005) 280-294
13. McAllister, A.: Complete rules for n-ary relationship cardinality constraints. Data Knowl. Eng. 27 (1998) 255-288
14. OMG: UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14) (2003)
15. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
16. OMG: MDA Guide Version 1.0.1. (2003)
17. OMG: XML Metadata Interchange (XMI) Specification v.2.0. OMG Adopted Specification (formal/03-05-02) (2003)
18. OMG: Revised submission for MOF 2.0 Query/Views/Transformations RFP. (ad/2005-03-02) (2005)
19. Sunyé, G., Pollet, D., Traon, Y. L., Jézéquel, J.-M.: Refactoring UML Models. In: Proc. 4th Int. Conf. on the Unified Modeling Language (UML'01), LNCS, 2185 (2001) 134-148